Integrating Z and Cleanroom

Allan M. Stavely
Computer Science Department
New Mexico Tech

Abstract

We describe an approach to integrating the Z specification notation into Cleanroom-style specification and verification. In a previous attempt, a group at IBM used formal refinement of the Z in their development. They concluded that this was not cost-effective in a commercial environment, and the attempt was not judged successful. The current approach avoids formal refinement, and instead begins by converting the Z to a fully constructive form, expressing all state changes using an assignment notation. The development then proceeds in Cleanroom style, with sections of the Z specification simply distributed among the program components to serve as their specifications. In a pilot project, this approach was found to work quite well, with development proceeding smoothly and predictably as normally expected with Cleanroom methods.

1 History of the problem

In the early 1990s, a group of technical staff at the IBM laboratory at Hursley Park (near Winchester, England) attempted to integrate two software engineering technologies which IBM had previously used separately with considerable success: the Z specification notation and the Cleanroom method.

The Z notation [15] [6] [13] [17] [18] is based on set theory and other basic elements of discrete mathematics, and incorporates novel structuring constructs (schemas and the schema calculus). Z technology also includes methods for the formal refinement of specifications into designs and code.

The core of the Cleanroom method [10] [8] [16] is formal or semiformal specification, and corresponding verification done by a development group in review meetings. Other elements of the method include notations and techniques for stepwise refinement, testing based on expected usage patterns, statistical analysis of test results to predict product quality, and incremental development.

IBM had had considerable experience with both technologies. The Cleanroom method was developed largely at IBM, by Harlan Mills and his colleagues in the Federal Systems Division. By the time of the Hursley experiment, it had been used successfully on a number of industrial-sized projects at IBM and elsewhere. The results were striking: very low levels of defects in the products, with no net loss and often a net gain in productivity [8] [3].

IBM had just finished a substantial development project at Hursley using Z, in collaboration with its developers at Oxford University [5]. The project was a major new release of the CICS transaction processing system: 268,000 lines of new and modified code, of which 37,000 lines were specified and designed using Z and another 11,000 lines were partially specified in Z. For the parts produced using Z, IBM reported a higher percentage of defects eliminated early in the development, a lower level of defects in the final product, and an estimated 9% reduction in development costs. IBM and Oxford were jointly given the Queen's Award for Technological Achievement for 1992 on the basis of this work.

The CICS group at Hursley hoped that Z and Cleanroom methods could be used together, and would complement each other to produce products of even higher quality than with either separately. The approach that they took was to write specifications in Z initially; to proceed with formal refinement steps as normally done in Z; to write the correctness criteria for these refinements as mathematical theorems; and to prove these theorems in review meetings, as normally done in Cleanroom.

The experiment was not judged a success. In particular, the group found it too hard to do the formal refinement from Z into code. The postmortem [12] concluded that "it is not cost-effective in a commercial software environment to do even semi-formal refinement without machine assistance" (which was not available).

Despite this discouraging result, we felt that there

was much to be gained if Z and Cleanroom methods could be integrated successfully. In the following, we describe a quite different approach. We avoid formal refinement in Z altogether, and instead begin by translating Z specifications into a form that more closely resembles Cleanroom-style specifications. From there, the development proceeds in Cleanroom style, but retaining fragments of Z notation where appropriate. We found that, using this approach, the Z notation can complement Cleanroom methods quite effectively.

2 Z and Cleanroom specification styles

The Z notation is well suited to expressing the specification of a system as a whole, or of major parts of a system. It provides a great deal of useful mathematical vocabulary, and the vocabulary of discrete mathematics in particular, which can be used very effectively to specify aspects of an information-processing system at a high level. Furthermore, it provides the schema notation and the schema calculus, by means of which many different aspects of a specification, each perhaps derived from a different requirement of the system, can be expressed separately and then combined into a single specification.

The Cleanroom method, on the other hand, provides relatively little built-in notation. Indeed, one of its strengths is that many kinds of notation, from a wide variety of domains and at many levels of formality, can be imported into it and used in its specifications. What it does provide is, in particular, a straightforward method of placing specifications on the lower-level components of a program, down to the level of the control construct or statement, and verifying that these components satisfy their specifications.

It would seem to be a natural idea, then, to begin by writing the top-level specification of a system using Z, and then to proceed with the development in Cleanroom fashion, distributing the Z specification among the program components and verifying those components against the specification fragments using Cleanroom protocols in review meetings.

However, there is a gap that must be bridged before the Z notation can be incorporated into Cleanroom-style specifications. This is because there are fundamental differences in the styles of the specifications of Z and Cleanroom.

The Z notation is based on predicates, which express preconditions and postconditions on operations, invariants on data, and other assertions and constraints on the data objects and inputs and outputs of a system. In particular, the specification of an operation defines a relation among inputs, outputs, previous values of state variables, and new values of those variables.

A fundamental property of Z is that such specifications may be *nonconstructive:* they may express properties that outputs and new values of variables must satisfy, without giving any clue as to how these values can be calculated from inputs and previous values of variables. In fact, specifications may even be *nondeterministic:* they may not constrain each output and updated variable to a unique value.

Here is an example which is both nonconstructive and nondeterministic, from the specification of a text-processing system: [6, p. 172]:

$$[CHAR]$$
$$TEXT == \operatorname{seq} CHAR$$

Format

$$t, t' : TEXT$$

$$words\ t' = words\ t$$
$$\forall l : \operatorname{ran} (lines\ t') \bullet \#l \leq width$$

In the specification of an operation in Z, the name of a state variable is "decorated" with a $'$ symbol to refer to its new value; the undecorated variable name refers to its previous value. (Input variables are decorated with ? and output variables with !.) Thus, this schema says that a *Format* operation leaves the sequence of words in t unchanged and that each line of t after the operation must be no longer than *width* (the functions *words* and *lines* and the constant *width* are defined elsewhere). The specification says nothing about how to achieve this result and, in fact, there will usually be many ways of dividing t into lines that will satisfy this specification.

Z practitioners see the ability to write nonconstructive and nondeterministic specifications as an advantage:

> Non-deterministic operations are important because they sometimes allow specifications to be made simpler and more abstract [15, p. 131].

> Nonconstructive specifications achieve expressivity and brevity at the expense of executability . . . they leave the programmer free to choose among different implementation strategies [6, p. 38].

In the Cleanroom method, on the other hand, the general rule is that specifications are both deterministic and constructive. Specifications are written in the "functional" style [9], in which each operation, control construct and statement in a program is viewed as computing a function on the program's state:

$$X := f(X)$$

Here X is a state vector that encompasses all of the program's state variables, including its input and output streams. Specifications are written in the form of *intended functions* which explicitly give values for every state variable which changes value. The usual notation is the *concurrent assignment*, such as the following:

$$[\ sum,\ i,\ trend :=\\ sum + a[i],\ i+1,\ (sum + a[i])/(i+1)\ ]$$

A variant is the *conditional concurrent assignment*, which specifies a state change by cases, such as the following:

$$[\ i > 0\ \rightarrow\ trend := sum/i\\ \mid\ i = 0\ \rightarrow\ sum, trend := 0, trend_0\ ]$$

Each case has a precondition and a concurrent assignment which is the state change to be performed when the precondition is satisfied; the computation is undefined whenever no precondition is satisfied.

The usual situation is that the preconditions of a conditional concurrent assignments are mutually exclusive (there is no state in which any two are both true) and that the right-hand side of each concurrent assignment contains only single-valued expressions which are obviously computable. In this case, the specification is deterministic and constructive. Exceptions are occasionally made, and occasionally a specifier will depart from this notation entirely. However, the rest of the Cleanroom method, and the verification in particular, will usually proceed more smoothly if the above conventions are followed. One reason for this is that a common manipulation in verification is to substitute the result of a computation into the specification of a following computation and then simplify.

3 The transition from Z to Cleanroom

The first step in our adaption of a Z specification to Cleanroom-style development and verification, then, is to transform the Z into a deterministic and constructive specification, so that it can be expressed using the intended functions required by the Cleanroom method. It might seem that this would be a nontrivial task, requiring a great deal of effort and introducing many opportunities to make mistakes that will jeopardize the success of the project.

However, in our experience thus far, we find that the job is usually not as hard as one might think. This is largely because many parts of typical Z specifications are already deterministic and constructive. In particular, we find that many Z predicates are of the form

$$v_1 = e_1 \wedge v_2 = e_2 \wedge \ldots$$

or

$$P \wedge v_1 = e_1 \wedge v_2 = e_2 \wedge \ldots$$

or

$$(P_1 \wedge v_{11} = e_{11} \wedge v_{12} = e_{12} \wedge \ldots) \vee\\ (P_2 \wedge v_{21} = e_{21} \wedge v_{22} = e_{22} \wedge \ldots) \vee \ldots$$

where each v is a changed state variable or an output variable (i.e., an variable decorated with $'$ or !) and such variables do not occur in any P or e, and where (in the third form) the P_i are mutually exclusive. Such predicates define computations that are clearly both deterministic and constructive, assuming that each P and e is single-valued and there is an obvious way to compute it. Furthermore, it is trivially easy to rewrite any such predicate in conditional-concurrent-assignment form. In fact, they are essentially in that form already, except for the symbols used.

Fortunately, such forms are natural to use in many situations in Z specifications, and Z users seem to use them rather commonly. In 28 case studies presented in six prominent Z books [15, ch. 1] [4, parts B–D] [6, ch. 16–25] [13, ch. 8] [17, ch. 15 and 20–23] [18, appendix A], over 67% of the 353 schemas which imply state changes or output are already in one of the above forms, once the schemas that are defined by including or combining other schemas are expanded out into their full forms. Another 6% contain instances of (for example) the new value of one variable being defined in terms of the new value of another, in contexts like

$$a' = f(a) \wedge\\ b' = g(a')$$

in which the departure from the above forms can easily be eliminated by an obvious substitution. Again, the translation to conditional-concurrent-assignment form is straightforward.

We could proceed, then, by translating all of the specifications of operations directly from Z predicates to conditional concurrent assignments, routinely in the easy cases and using more complex transformations in the other cases. However, to make the transition smoother, we devised an intermediate form which combines characteristics of both notations. It is very much like standard Z — in fact, it can be considered a nonstandard dialect of Z — except that all state changes are specified explicitly and constructively.

Here are the principal differences between this notation and standard Z.

- State changes are written in the form

$$x := E$$

This is equivalent to the standard Z

$$x' = E$$

but the change in notation emphasizes the explicit, constructive definition of the state change. The same assignment notation is used to specify the computation of outputs.

- Every change to a state component is specified in this way; it is implied that no other state component changes its value. With this convention, all assertions of the form

  $$x' = x$$

  are omitted as redundant from schemas that specify state changes.

  There are no implicit changes to one state component induced by changes to another state component and constraints between them. All state changes are written out explicitly.

- In the same spirit, where it is asserted that part of a structured state component is changed, it is implicit that the rest of the component remains unchanged. In particular, where the state component is a mapping (in Z represented as a function), a change to its value on one element of its domain can be written in the form

  $$f\ a := E$$

  If this is the only change to f that is specified in the schema in which this appears, it is implied that f remains unchanged otherwise, and the above is equivalent to the standard Z

  $$f' = f \oplus \{ a \mapsto E \}$$

  where $\oplus$ is the "override" operator.

  More than one change to the same function can be specified:

  $$f\ a_1 := E_1 \land f\ a_2 := E_2$$

  means

  $$f' = f \oplus \{ a_1 \mapsto E_1, a_2 \mapsto E_2 \}$$

  which, of course, is well-defined (i.e., f' is still a function) only if $a_1 \neq a_2$ or $E_1 = E_2$.

  A change to a (curried) function of two arguments can be written as

  $$f\ a\ b := E$$

  which (if no other changes to f are specified in the schema in which this appears) is equivalent to

  $$f' = f \oplus \{ a \mapsto ((f\ a) \oplus \{ b \mapsto E \}) \}$$

  and so on for functions of more arguments.

- Since the syntax $x := E$ is really a predicate, it can appear anywhere a predicate can appear, such as within the scope of a quantifier. An example is

  $$\forall x : T \mid x \in S \bullet \\ \quad f\ x := a$$

  which means

  $$f' = f \oplus \{ x : T \mid x \in S \bullet x \mapsto a \}$$

- The symbols Δ and Ξ are now superfluous in most places and may be omitted.

- All computations of new states and outputs appear only in contexts which are unconditional, or in conditional structures (using $\lor$ and $\land$) with mutually exclusive conditions.

Many of these notation conventions are similar to constructs in the notation of the B method [19], although that notation is more restrictive than AZ in a number of ways.

Since state changes are specified in the form of assignments, we tentatively call this variant of the Z notation "Assignment Z", or AZ. (We considered the name "Constructive Z", but this name is already in use with a somewhat different meaning [11].) We present AZ not as another formal specification notation, but merely as an informally-defined intermediate form between Z and conditional concurrent assignments.

Where the Z specifications are not already constructive, we transform them into a constructive form as we rewrite them in AZ notation. For example, we would rewrite

Pop

$stack, stack' : \mathrm{seq}\ Item$
$x! : Item$

$stack = \langle x! \rangle \frown stack'$

as

Pop

$stack : \operatorname{seq} Item$
$x! : Item$

$x! := head\ stack$
$stack := tail\ stack$

Often, as here, making a state change constructive is rather easy, but it can require considerable manipulation.

There is sometimes more than one way to express the constructive version, and whatever choice is made will usually suggest a design or implementation possibility more strongly than the nonconstructive version did. Similarly, where the specification is nondeterministic, making it deterministic typically involves either making arbitrary choices as to the result that is specified, or making choices influenced by design or implementation considerations. An example is an allocation of a resource from a set of numbered resources:

Allocate

$free : \mathbb{F}\,\mathbb{N}$
$allocated' : \mathbb{N}$

$allocated' \in free$
$free' = free \setminus allocated'$

(Here $\mathbb{F}$ means "finite set of" and $\mathbb{N}$ denotes the natural numbers.) As we convert this to AZ form, we might make it deterministic by arbitrarily choosing the free resource with the minimum number:

Allocate

$free : \mathbb{F}\,\mathbb{N}$
$allocated : \mathbb{N}$

$allocated := min\ free$
$free := free \setminus \{min\ free\}$

Another way of resolving the nondeterminism would be to define *free* to be a sequence rather than a set, and choosing the first element of the sequence every time:

Allocate

$free : \operatorname{seq} \mathbb{N}$
$allocated : \mathbb{N}$

$allocated := head\ free$
$free := tail\ free$

Clearly, this version encourages a different implementation. It is important to realize that the transformations that we perform to make the specification constructive and deterministic are not just changes in notation, but are true development steps, and may involve nontrivial and significant design decisions.

It is probably not necessary to be too dogmatic about removing all nonconstructive and nondeterministic aspects of the specification at this stage. Consider this example:

DisplayPeople

$knownPeople : \mathbb{F}\ PERSON$
$people! : \mathbb{F}\ PERSON$

$people! = knownPeople$

This is reasonable Z, but of course if a set is displayed as an output, it must be displayed in some order. A possible conversion to AZ might be:

DisplayPeople

$knownPeople : \mathbb{F}\ PERSON$
$people! : \operatorname{seq} PERSON$

$people! = alphabeticalSort\ knownPeople$

where *alphabeticalSort* denotes sorting by name in phone-book order. One might object that this is still both nonconstructive and nondeterministic, since it does not suggest how the sorting is to be done, and it may allow more than one outcome if more than one person can have the same name. But any other way of writing this specification is likely to be more complicated and less satisfactory. Furthermore, it is obvious that any competent programmer can create an implementation that will satisfy it. In such situations the pragmatic thing to do may be to allow specifications such as this one, although we should do so only after careful consideration.

4 A pilot project

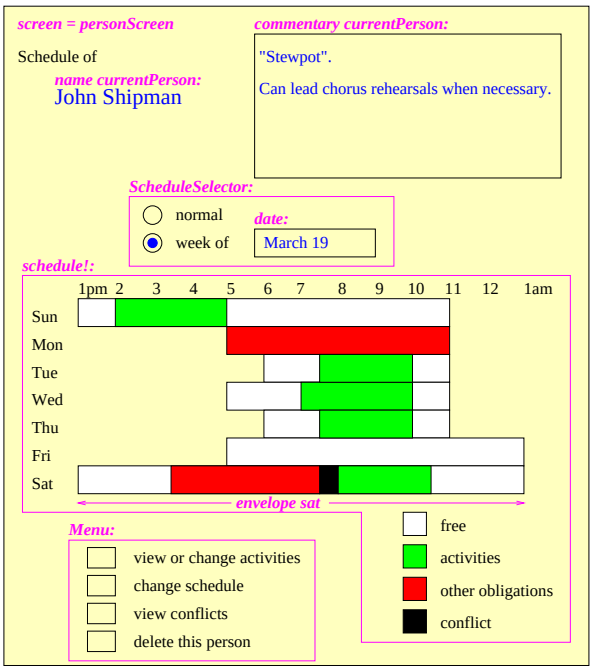
To try out the techniques presented above, we attempted use them on a development project of modest size. As it happened, we had a project that we were already planning to undertake.

The project was to develop a "rehearsal scheduler's assistant": a program to help with the planning and scheduling of the rehearsals and other preparation for a theatrical production. The central job to be done is to manage the interacting schedules of many activities and many people. We had a real client, Doug Dunston, the faculty member in charge of the music program in our college. (The project is described in somewhat fictionalized form in section 11.3 of [16].)

The first step of the development, after discussing requirements with the client, was to prepare a specification

in standard Z. As is common practice with Z, the specification took the form of a document, with sections of Z interspersed with explanatory text in English. The specification contained 43 schemas, and 16 other Z sections containing definitions of various kinds.

The specification document contained one other important specification notation: color pictures of the screens and other components of the graphical user interface (GUI). Here is an example:



We used pictures like these to include in the specification a general idea of what the GUI would look like. However, we adopted the convention that the pictures would represent only an approximation to the appearance of the interface, which might vary somewhat according to the eventual implementation. Colors and dimensions (for example) might be slightly different from the way they appear in the pictures, and there might be implementation-dependent features not shown in the pictures, such as additional ways to move from one screen to another.

On the other hand, some aspects of the pictures are quite specific. In particular, some of the elements of each picture are tied to the Z specifications through labeling conventions. In the picture, an annotation of the form ***someName:*** (which appears in a distinguished color, magenta, when the specification is printed or displayed in color) is not to appear in the actual GUI as displayed, but indicates that the corresponding part of the GUI corresponds to the construct of the same name in the Z specification. If there might be any doubt as to what part of the display is being referred to, a box of the same color is drawn around the relevant part; again, this will not actually appear in the GUI.

Here is the Z schema that corresponds to the above picture:

$$
\begin{array}{|l}
\hline
\textit{PersonScreen} \\
\textit{PersonData} \\
\textit{ScheduleSelector} \\
\textit{PersonScheduleDisplay} \\
\textit{Menu}[\textit{SCREEN}] \\
\hline
\textit{screen} = \textit{personScreen} \\
\textit{preselected}! := \textit{normal} \\
\textit{date} = \textit{today} \\
\textit{menuChoices}! := \\
\quad \{\, \textit{editPersonScreen} \mapsto \\
\qquad \text{“view or change activities”}, \\
\quad \textit{personScheduleScreen} \mapsto \\
\qquad \text{“change schedule”}, \\
\quad \textit{showConflictsScreen} \mapsto \\
\qquad \text{“view conflicts”}, \\
\quad \textit{deletePersonScreen} \mapsto \\
\qquad \text{“delete this person”} \,\} \\
\textit{screen} := \textit{chosenItem}? \\
\hline
\end{array}
$$

The annotation ***screen = personScreen*** in the picture indicates that *screen* has the value *personScreen* when what the user sees is the screen shown in the picture. The variables *date* and *preselected*! are defined in the schema *ScheduleSelector*; *preselected*! defines which of the two selector buttons is initially shown as selected. Whenever a schema defines a GUI component, we define informally in the accompanying text how the Z components relate to what the user sees and can manipulate.

We adopted several other conventions to reduce the amount of repetitive detail in the specification. For example, in many places in the GUI there is a box in which the user can fill in or edit a value. Wherever a picture contains such a box labeled with the name (for example) *x*, and *x* is a variable of type *T*, we treat that annotation as implicitly introducing a Z schema of the form

$$
\begin{array}{|l}
\hline
\textit{Edit\_x} \\
\textit{displayed\_x}! : \textit{TEXT} \\
\textit{entered\_x}? : \textit{TEXT} \\
x : T \\
\hline
\textit{displayed\_x}! := \textit{TtoTEXT}\ x \\
x := \textit{TEXTtoT}\ \textit{entered\_x}? \\
\hline
\end{array}
$$

where *TtoTEXT* and *TEXTtoT* are appropriate conversion functions. The box labeled ***date:*** is an example

of this; the schema *personScreen* also specifies that the value initially displayed for *date* is today's date.

The specification document, then, contains English text, Z notation, and pictures, all interrelated. It should be apparent that some parts of the specification are formal and other parts are quite informal. In all, the document is 42 pages long.

The next step, after meeting again with the client to discuss that specification and obtain his approval, was to prepare another version of the document in which the parts of the Z sections were rewritten in AZ form. This turned out to be quite easy in most places, especially since most of the state changes were specified in such a way that the translation was trivial, as discussed in the previous section. In many cases, the resulting specifications turned out to be considerably simpler than the original, largely because of the AZ convention for expressing changes to components of structured objects. For example, the specification used curried functions like the following in many key places:

$$
\begin{array}{l}
SCHEDULESTATUS ::= \\
\quad free \mid booked \mid otherObligations \mid conflict \\
DAYSCHEDULE == \\
\quad TIME \pfun SCHEDULESTATUS \\
WEEKSCHEDULE == \\
\quad DAYOFWEEK \to DAYSCHEDULE
\end{array}
$$

NormalSchedules

$$
\begin{array}{l}
\ldots \\
People \\
normalSchedule : \\
\quad PERSON \pfun WEEKSCHEDULE \\
\hline
\ldots
\end{array}
$$

This was a natural way of constructing *normalSchedule*, especially since we sometimes wanted to refer to the whole weekly schedule of a person, sometimes for that schedule on a particular day, and sometimes to *that* schedule at a particular time. But then specifications of state changes like the following became complex and tedious:

$$
\begin{array}{l}
normalSchedule' = normalSchedule \oplus \\
\quad \{ currentPerson \mapsto \\
\quad (normalSchedule\ currentPerson) \oplus \\
\qquad \{ day \mapsto \\
\qquad (normalSchedule\ currentPerson\ day) \oplus \\
\qquad\quad \{ t \in possibleTimes\ day \mid \\
\qquad\qquad\qquad from \leq t < to \bullet \\
\qquad\qquad t \mapsto selected? \} \} \}
\end{array}
$$

The AZ form of this is much more straightforward:

$$
\begin{array}{l}
\forall t \in possibleTimes\ day \mid from \leq t < to \bullet \\
\quad normalSchedule\ currentPerson\ day\ t \\
\qquad := selected?
\end{array}
$$

In determining what needed to be rewritten to make it constructive, we were guided by pragmatic considerations. For example, the original specification contained a number of state changes specified using set comprehensions, in forms such as

$$result := \{a \in S \mid P(a)\}$$

But in each such case, S was a finite set and so, in principle at least, the set of its elements satisfying P could be constructed by a simple-minded enumeration of the set, testing each element. Indeed, for this reason a mathematician would probably consider such an expression quite constructive, and we judged all such specifications to be "constructive enough" for our purposes. In fact, in the implementation, each such set turned out to be reasonably small, and so this is exactly how almost every such state change was actually implemented.

For a number of reasons, including portability (the program was to be developed on our Linux machines but would eventually run on Dr. Dunston's Macintosh), we chose the Python programming language and the Tkinter GUI library for the implementation.

We found it easy to implement many parts of the AZ specification using Python constructs, in ways that so obviously matched the specification that verification was hardly necessary. This was especially true of state changes that called for modifying values of functions. We implemented the function *normalSchedule*, for example, as a dictionary indexed by *Person* and *Activity* objects, containing lists indexed by numbers representing days of the week, where those lists contained dictionaries indexed by *Time* objects and containing *ScheduleStatus* objects. Thus, for example, the implementation of the state change specified by

$$
\begin{array}{l}
\forall t \in possibleTimes\ day \mid from \leq t < to \bullet \\
\quad normalSchedule\ currentPerson\ day\ t \\
\qquad := selected?
\end{array}
$$

turned out to be simply

```
for t in possibleTimes(day):
   if fromTime <= t < toTime:
      normalSchedule[currentPerson] \
            [day][t] \
         = selected
```

which is essentially identical to the specification.

We used one other significant piece of software engineering technology in the project: a form of "literate

programming" [7]. This means that the program is prepared and presented in the form of a document, with explanatory text accompanying each section of program code. Thus the program and its documentation are integrated, and stored in a single file. There are software tools that process that file either to strip out and order the code sections for compilation and execution, or to format the document for viewing or printing.

In the usual kind of literate programming, the code fragments may appear in the document in any order, but the author must use markup commands to define their ordering and nested structure in the final program. We adopted a much more "lightweight" approach, in which the code fragments appear in the program in the same order in which they are presented in the document. This is reasonable with Python, in which the order of elements in a program is relatively unconstrained. And it means that the program that strips out the code fragments for execution does not need to do any reordering, which made it a very simple program. Even more important, it means that the programmer does not need to include markup to structure the program fragments. In fact, the only extra markup necessary is a pair of commands defined in the markup language (LaTeX in our case) to mark the beginning and end of a code fragment.

With very little extra effort, then, we were able to maintain the code and a description of it as a readable document. In many places we also included sections of Z and pictures of the GUI, cut-and-pasted from the specification documents. The result is a document very similar to those documents in style and appearance, but with code sections added. Each code section is accompanied by commentary and, in many cases, the Z and pictorial specifications from which the code was derived.

In many places we also extracted fragments of Z from the specification and incorporated them into intended functions in the code fragments. Here is an example:

V

```
def emptyDaySchedule(d):
[ returned value := a dictionary
      { t ∈ possibleTimes(d) • t ↦ free } ]
   result = { }
   for t in possibleTimes(d):
      result[t] = free
   return result
```

By the way, the annotation "**V**" in the corner of the box indicates that this section of code has been verified. An annotation "**VT**" would indicate that it had also been tested. We used such annotations to keep track of the status of each fragment directly in the program document during the development, and we found this very helpful.

We did not follow Z protocols for formal refinement at all. The implementation was constructed using ordinary programming skills, as well as Cleanroom-based methods in which intended functions are implemented in a stepwise manner in terms of code and lower-level intended functions [16, ch. 5]. We constantly used the structure of the Z specification as a guide, and this made many parts of the implementation almost trivial to construct.

We verified both the translation of the Z to AZ, and the program code developed from the AZ. Unfortunately, we were unable to adhere strictly to Cleanroom methods in doing this. Cleanroom is inherently a group process; in particular, verification is done in review meetings, with the author and colleagues discussing each correctness criterion and examining the program for other aspects of quality. The goal is to discover and eliminate as many defects as possible while attempting the verification. Normally this requires a group of at least three people, since each person often notices defects that the others miss.

But only one other person trained in Cleanroom methods was available at the time, and the amount of time that he had available was quite limited. Therefore, parts of the program were verified in a two-person group, and some parts were done strictly as a solo effort. We found that verification under these circumstances was far less reliable than normally expected with Cleanroom methods, which typically achieve a level of defects of three per thousand lines of code or better before first execution [8] [3].

To add to our difficulties, we were somewhat unfamiliar with the Python language and the Tkinter library, and made a number of minor mistakes in usage, especially early in the project. Since Python is an interpreted language, the mistakes that escaped our notice during verification were not caught in compilation, but in first execution.

To attempt to compensate, we eventually developed an alternate protocol. The project plan called for the program to be developed in rather substantial increments, as is normal with the Cleanroom method. We divided each of these into a number of very small increments, each adding perhaps only one simple new feature to the program; these increments ranged in size from about thirty to two hundred new and changed lines.

After each of these increments was coded, we inspected it several times, using a checklist and checking different aspects each time. We checked such things as points of syntax and usage which had caused us prob-

lems before, matching of each function and method call against its definition (comparing both intended functions and number and types of parameters), and correspondence of intended functions with the Z in the specification document. Finally we inspected for correctness of each section of code with its intended functions. In some cases, as in the *normalSchedule* example above, we judged the code obviously correct "by inspection"; in other cases we carried through more detailed correctness arguments, mentally or on paper [16]. We caught and eliminated many defects by means of these inspections, about four defects per hundred lines on average.

Each increment was then integrated into the program; thus we were, in effect, "growing" the program gradually, as advocated by Brooks [1, p. 18]. At each integration step we ran a few cursory tests to execute each new piece of code for the first time, and many more defects showed up immediately. The defect density on first execution was about five per hundred lines on average, not nearly as good as normally expected with Cleanroom methods. Thus, our one-person inspection protocol does not come close to competing with a full Cleanroom-style verification review by this measure.

Fortunately, this had little effect on the effectiveness of the development! Almost all of the defects that survived the inspections were caught on first execution and were simple oversights: typographical errors, mistakes in punctuation, mistakes in names of variables, omitted initialization, and the like. Each took only a few minutes to track down and fix. There were no deep algorithm flaws, no subtle bugs which would cause malfunctions only rarely, and no places in which we had implemented algorithms that would do something quite different from what was specified. This is typical of what normally happens in a Cleanroom-style development: the really nasty bugs are the ones that specification and verification seem to be most effective at preventing or eliminating.

Most important for the subject of this paper, the entire development went very smoothly. At no time did we feel that the mixture of notation was a hindrance or added excessive complexity to the process. On the contrary, we felt that it was definitely helpful to have a Z specification to use a a basis for the development, and that specifying the program using the vocabulary of discrete mathematics right from the beginning probably made the design cleaner than it would otherwise have been. We also felt that using Cleanroom-style intended functions and stepwise refinement definitely contributed to the quality of the product, as did inspections, imperfect as the latter were. These are subjective judgments for the most part, of course, but we think they are justified.

At the time of writing, the third of the five major increments called for in the project plan has been completed, resulting in 1409 nonblank, non-comment lines of code in the Python language. (We estimate that several times as many lines would have been needed in a lower-level language such as C or Java.) We found only five defects in further testing; this means that the defect density that we obtained after inspection and first test during integration is comparable to the defect density normally obtained after verification in Cleanroom.

Dr. Dunston has begun to use the program experimentally, and intends to put it into full production use for his next musical comedy. By that time the remaining increments will be constructed and installed. Meanwhile, we have begun to use the program in our own work, to help schedule the activities of the staff of an introductory computer science course (lecturers, teaching assistants, tutors and graders) around all of their other obligations. The program has been quite helpful with this. As of the time of writing, no further defects have been found in the program.

5 Conclusions

We consider that the integration of Z and Cleanroom, as described above, was successful. We believe that the use of specification via pictures and of "lightweight" literate programming contributed to the success of the project as well. Results obtained from one project of this size are not conclusive, of course, but all indications are positive thus far.

We definitely intend to use similar combinations of technologies in future projects, and are eager to try them on substantially larger projects. Since Z and Cleanroom have been used separately on projects of substantial size with considerable success, we see no reason why the same should not be true when they are used together, but only actual experience will tell us with certainty.

Beyond this, we believe that our results confirm and support several ideas already noted by other writers and researchers regarding the way to use formal methods most effectively. First, formal methods are not monolithic: it is quite possible to use some parts or aspects of a method without using all of the method. For example, it makes perfect sense to write specifications in Z even if one has no intention of using the accompanying methods for formal refinement, and doing this seems to be rather common among Z users.

Similarly, it is perfectly reasonable to use more than one formal method or notation in a project, according to which is most suitable for each part of the project. A notable example of this was the development project for the CDIS air traffic control display system [2], which successfully used a variety of formal notations: VDM, VVSL, CSP and CCS, as well as data-flow diagrams and

finite-state machines.

Finally, full formality is not only not necessary to obtain the benefits of formal methods, but is frequently not even productive or cost-effective. In the postmortem to the Hursley experiment [12, p. 293], Mark Pleszkoch of the IBM Cleanroom Software Technology Center is quoted as saying:

> I believe that the key to applying Cleanroom in a cost-effective, highly productive manner is to not force developers to go to a level of formality beyond their needs (and abilities), while at the same time not losing the benefits of precise documentation that makes clear what each piece of code is designed to do.

A number of other writers have been expressing similar opinions in recent years (see, e.g., [14] and [2, pp. 74–75]). The general principle is that there is an appropriate level of formality for every situation, and more rigor is not always better. If this is not yet the consensus of the formal methods community, perhaps it eventually will be.

Acknowledgement: We are indebted to Steve Powell of IBM at Hursley for many thoughtful comments and suggestions.

References

[1] Frederick P. Brooks, Jr. "No silver bullet: Essence and accidents of software engineering." *Computer 20*, 4, pp. 10–19, April 1987.

[2] Anthony Hall. "Using formal methods to develop an ATC information system." *IEEE Software 13*, 2 (March 1996), pp. 66–76.

[3] P. A. Hausler, R. C. Linger and C. J. Trammell. "Adopting Cleanroom software engineering with a phased approach." *IBM Systems Journal 33*, 1 (1994), pp. 89–109.

[4] Ian Hayes, ed. *Specification Case Studies*. London: Prentice Hall International (UK) Ltd, 1987.

[5] Iain Houston and Steve King. "CICS project report: Experiences and results from the use of Z in IBM." In *VDM '91: Formal Software Development Methods*, pp. 588–596. Berlin: Springer-Verlag, 1991.

[6] Jonathan Jacky. *The Way of Z*. Cambridge, England: Cambridge University Press, 1997.

[7] Donald E. Knuth. "Literate programming." *The Computer Journal 27*, 2 (May 1984), pp. 97–111.

[8] Richard C. Linger. "Cleanroom process model." *IEEE Software 11*, 2 (March 1994), pp. 50–58.

[9] Harlan D. Mills. "The new math of computer programming." *Commun. ACM 18*, 1 (January 1975), pp. 43–48.

[10] Harlan D. Mills, Michael Dyer and Richard C. Linger. "Cleanroom software engineering." *IEEE Software 4*, 5 (September 1987), pp. 19–24.

[11] Seyed-Hassan Mirian-Hosseinabadi and Raymond Turner. "Constructive Z." *J. Logic Computat. 7*, 96-48 (1997), pp. 49–70.

[12] Glyn Normington. "Cleanroom and Z." In *Z User Workshop, London 1992*. London: Springer-Verlag, 1992.

[13] Ben Potter, Jane Sinclair and David Till. *An Introduction to Formal Specification and Z* (second edition). Hemel Hempstead, England: Prentice Hall Europe, 1996.

[14] Hossein Saiedian, ed. "An invitation to formal methods." *Computer 29*, 4 (April 1996), pp. 16–30. See particularly the contributions of Jones, Jackson and Wing, and Lutz.

[15] J. M. Spivey. *The Z Notation: A Reference Manual* (second edition). Hemel Hempstead, England: Prentice Hall International (UK) Limited, 1992.

[16] Allan M. Stavely. *Toward Zero-Defect Programming*. Reading, Mass.: Addison Wesley Longman, 1999.

[17] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement and Proof*. Hemel Hempstead, England: Prentice Hall Europe, 1996.

[18] J. B. Wordworth. *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*. Wokingham, England: Addison Wesley, 1992.

[19] J. B. Wordworth. *Software Development with B: An Introduction*. Harlow, England: Addison Wesley Longman Limited, 1996.